Applications

HABA HIPPO-C™

FOR ATARI ST™

Table of Contents

Acknowledgements

We at Haba Systems and Hippopotamus Software gratefully acknowledge the following people who contributed their time and effort to the creation of Haba Hippo-C. Clint Ballard acted as chief technical guru, while Wendell Brown directed the company and coordinated the entire project. Many thanks to our beta-testers and others for their valuable suggestions and input. We sincerely appreciate the support and encouragement from everyone at Atari Corporation.

Chapter 1: Introduction

Welcome! Haba Hippo-C is a pleasant, interactive C environment which allows you to easily edit, compile, assemble, link, and run C programs on the Atari ST series computers.

The entire Haba Hippo-C system has been designed into an integrated environment which allows maximum programming efficiency. Batch files allow you to easily automate the entire process of compiling, assembling, linking, and running C programs.

The **Help** key on the keyboard offers you an on-line summary of the many useful commands available when you're in HOS or the editor.

The Haba Hippo-C compiler follows the standards, excluding floating point, defined in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. We highly recommend that you use that book for reference when programming with Hippo-C.

Access to the Atari's GEMDOS, GEM VDI and AES routines is convenient from Hippo-C. Simply call them as you would any other C routine. A partial description of the binding routines are included in this documentation. However, we strongly recommend that you obtain a copy of the GEMDOS specification. Contact Digital Research, your local Atari dealer, or Atari Corporation for further information.

We would like to receive your comments, criticisms, suggestions, and improvements on how to make Haba Hippo-C even better. Your ideas will be greatly appreciated.

Loading Haba Hippo-C

If you have a two drive system, insert your TOS System disk into drive B and your Haba Hippo-C disk into drive A. If you have just one drive, insert your TOS disk into the drive. Once TOS boots up and the GEM desktop appears, eject the TOS disk and insert your Haba Hippo-C disk.

The first time you load Haba Hippo-C, you will have to copy the DESKTOP.INF file on the Haba Hippo-C disk to your TOS start-up disk. Drag the DESKTOP.INF icon to the icon of your TOS disk. If you have a one drive system, you may need to exchange disks a few times.

Once your system is set up, double-click the icon for drive A. in the window, you will see the HOS.PRG application icon. Open HOS.PRG by double-clicking on it.

Making Backup Copies

The Haba Hippo-C disk is copy protected. You may make a backup of your disk for your everyday use from the Desktop by dragging the Haba Hippo-C icon to the icon of a formatted disk. When you use the backup, **the original disk must be in drive A.** If you need a replacement of your Haba Hippo-C disk, contact Haba Systems.

Suggestions for Hard Disk Drive Users

Haba Hippo-C is compatible with Atari and compatible hard disk drives, like the Haba 10 Megabyte Drive. You should first move all files from your Haba Hippo-C disk to the hard disk. Then, you can compile and assemble from the hard disk.

Getting Help With Haba Hippo-C

This manual should answer all the questions you might have about using Haba Hippo-C. However, if you still need help with a particular feature after reading it, please call Haba Systems Technical Support at:

(818) 901-0701

This service is provided exclusively for registered Haba Systems customers who accept the terms of the License Agreement.

Chapter 2: Getting Started

Together, let's go through all the steps necessary to write a small C program, compile it, assemble it, link it, then run it.

1. If you're in the GEM window environment, double-click HOS.PRG to enter HOS (the Hippo-Operating-System). To return to GEM later, type **logout**.

Editing

2. Type **ed** to enter the editor.
3. Start by typing this program in:

```
main()
{
    printf("Hello world\n");
}
```

4. After you've typed in the program, press **F4** to save the file. You will be asked to name the file. Name it **foo.c** (the .c identifies the file as a C program). Hit the **Return** key to confirm the write.
5. Hit the **F10** key to quit and return to HOS.

Compile, Assemble, and Link

6. Now type **c foo** to compile and assemble the program. **C** is a batch file which automatically will type these HOS commands:

```
cc  foo.c
c2  foo.i
asm  foo.s
```


7. Now link **foo.o** by typing: **ld foo.o.**
8. Finally, you can run your program by simply typing: **a.out.** Your program will run, and you will be able to see the output appear on your screen.
9. **HabaHint:** As the compiler always calls the runable program **a.out**, you may want to rename the a.out file to another name using the **mv** command like this:

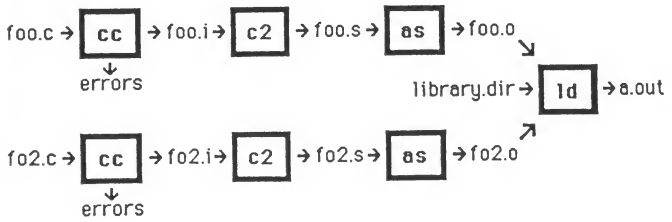
```
mv a.out myprogram
```

You can then simply run your program from HOS by simply typing the filename: **myprogram.**

The Entire Process

The entire process of converting a C program (often called a C source file) into a runable program involves the steps of compiling, assembling, and linking. You can compile and assemble many different C source modules, then link them all together using the **ld** (link) command. You can also link together 68000 assembly routines with C programs. The linker always produces an executable program called **a.out.**

THE ENTIRE PROCESS



It is possible to have different files with the same file name in different directories. Such as:

\bin\foo
\bin\oldstuff\foo
\usr\foo

In the example above, the filename **foo** is used in several directories, but each **foo** is still unique in content and path.

You can also have programs with different contents but with the same name. Thus, commands are first searched for in the current directory, then in the **\bin** folder. An executable file takes precedence over a batch file (**.bat**).

Wildcarding

Wildcarding allows you to easily specify a group of files without specifically typing each filename. For example, if you want to list the disk directory of all C source files, you could type:

ls *.c

The asterisk tells the utility program to accept any filename to the left of the period. Asterisks can be used any number of times in an argument. Wildcarding also allows using single or double quotes to isolate an argument.

The wildcard question mark (e.g. **ls foo?.c**) will match a file with that particular filename, allowing any one character to be substituted for the **?**.

Redirection of Input and Output

Standard input and output can be redirected by using the "<" and ">" characters. For example:

a.out > outfile

will invoke **a.out** and store its output in the file called **outfile.** Both input and output can be redirected simultaneously:

a.out < infile > outfile

In this case, the program **a.out** will get its input from "infile" and its output will go to **outfile.**

Command Keys

Several special keys are recognized by HOS:

control s - stops output to the screen (stdout or stderr)
control q - restart output stopped by option s
control d - EOF from keyboard
control c - program termination

Batch Files

Batch files (sometimes called "scripts" or "autotypers") save the programmer both time and effort. A batch file is essentially a normal text file comprised of a series of HOS commands. The same series of commands (often 5 to 10 lines) are usually repeatedly executed, so the batch file saves you from reentering the same series of command lines over and over again.

You can edit batch files just as you would any other file (e.g. **ed go.bat**). You can then type in your list of batch commands using the editor. In this case, once the batch file has been edited, you can execute that batch file from HOS by simply typing **go**.

You can pass arguments to batch files using '$0' thru '$9' as arguments. The batch file processor will parse out the batch command lines' arguments, then pass them through to your batch commands. For example, in the case of:

go thisfile

the sequence '$1' appearing anywhere in your batch file will temporarily be replaced by 'thisfile'.

Summary of HOS Commands

!!	--	retypes the most recently typed in command
A-P:	--	change to specified drive
ar	--	librarian, make 'library.dir' from 'archive.lis'
asm	--	assemble '.s' files and create '.o' files
c	--	batch to make a '.o' file from a '.c' file
c2	--	pass 2 of the C compiler, '.i' files to '.s' files
cat	--	concatenate files to standard output (to type)
cc	--	pass 1 of the C compiler, '.c' files to '.i' files
cd	--	change to a subdirectory
cp	--	copy one file
date	--	display or set date
df	--	display amount of disk space that is free
dump	--	hexadecimal dump of files
echo	--	echo arguments
ed	--	edit ascii file
help	--	display this message
ld	--	links '.o' files into 'a.out', -S for stand alone
logout	--	exit from HOS
ls	--	directory listing
mkdir	--	make subdirectories
mv	--	move (rename) one file
rm	--	remove files
rmdir	--	remove subdirectories
print	--	print file
pwd	--	display current subdirectory
time	--	display or set time
version	--	version number

Chapter 4: Program editing

The editor contains many convenient features that will help you write your program.

You can enter the editor from HOS by simply typing **ed**. To load in a file for editing, you must list it after the **ed** command (e.g. **ed foo.c**).

The editor is RAM based, which means that the file's entire contents must fit into available memory. This is not a serious limitation, since typically there is over 200K of available memory (about 2000 lines of typical C code). The editor **will not warn you** when memory is getting low, so it is best to limit your file size. We recommend that you limit each module's file size to less than 500 lines of C source code. You can, of course, link many modules together. This practice will not only prevent memory overflows, but also encourages a modular, structured programming style.

Moving Around the File

With the editor, you use the keyboard to move around the file. You can type, insert, delete, or replace text at the cursor. To move the cursor, press any of the four **arrow** keys. If you press the **Shift** key as you press an **arrow** key, you can move through your file more quickly. **Shift left arrow** moves the cursor to the beginning of the line, **Shift right arrow** moves the cursor to the end of the line, **Shift up arrow** moves the cursor up a page, and **Shift down arrow** moves the cursor down a page.

The **Clr/Home** key, located next to the arrow keys, can take you to the beginning or ending line of the file. Press **Clr/Home** to go the first line, or **Shift Clr/Home** to go to the last line of the file. By pressing **F5**, you can move to a specific line. Just enter the line number at the prompt and press **Return**, and the cursor will move to the beginning of the selected line. You can also move the cursor to the precise location you select. Move the cursor to the desired location and press **Esc**. When you want to move back to the point later, press **Shift F5**.

Editing

The editor works like your typewriter. Type your program lines the way you want. Indent your lines by pressing the **Tab** key, or use the **Space Bar** to insert spaces. When you reach the end of a line, press **Return**.

•Inserting Text

You can insert characters several ways with the editor. The easiest way to put the editor into the automatic insert mode. This means that you can move the cursor where you want the text inserted and start typing. The rest of the text will be pushed over. To start the automatic insert mode, press the **Insert** key. To return to the overstrike mode, press **Insert** again.

You can insert a single character into the text by moving the cursor to the desired location and press **F2**. If you want to insert an entire line, press **Shift F2**.

•Deleting Text

The easiest way to delete text is to press the **Backspace** key. This deletes the character to the left of the cursor. You can delete the character at the cursor by pressing **F1**. To delete the text from the cursor to the end of the line, press **Shift F1**.

•Cut, Copy, Paste and Delete

You can move or duplicate text in your file by cutting or copying it to a temporary holding area called the **clipboard** and pasting it in the desired location.

To select a block of text to move around, position the cursor with the **arrow** keys at one end of the block, and mark it using the **Esc** key. Move your cursor to the end of the block. If you want to remove the block entirely from its present location, press **Shift** and **F8**. If you want to copy the block to the clipboard, press **F8**. The block will remain in the clipboard until another block is cut or copied into it, or you quit the editor.

Move the cursor to where you want the block pasted and press **F9**. The text will be inserted at the cursor.

You can also delete a block of text completely from its present location without putting it on the clipboard. To do this, select the beginning of the block by pressing **Esc**, move the cursor to the end of the block by pressing the **arrow** keys, then press **Shift F9**. This will permanently erase the range, so be sure that is what you want to do before you press **Shift F9**.

•Search and Replace

The editor comes with a powerful search and replace routine which can instantly locate and change text. This is a handy tool for making changes or correcting errors throughout a file.

To search for a string, press **Shift F6**. A prompt at the top of the screen will ask you to enter the desired text. Type in the string and press **Return**. The cursor will move instantly to the first occurance of the text. To move to the next occurance, press **F6**. If the string can't be found, the prompt "String Not Found" will be displayed at the top of the screen. Press any key to remove it.

To search and replace a string, press **Shift F7**. Enter the text to be searched at the prompt, and press **Return**. You will then be asked to enter the replacement string. Enter it at the prompt at the top, and press **Return**. The first occurance of the search text will be found and replaced. To replace the text at the next occurance, press **F7**. If you just want to locate the next occurance without replacing it, press **F6**. You can also replace all instances of a string by pressing **Shift F3**.

•Showing the Status

You can find out how much memory you have left for your file by pressing **Shift Insert**. You will see a line like this:

```
FILENAME: X=10 Y=5 TOP=0 MAX LINES=1661 LAST=253
MEM=51898 REPLACE MODE MODIFIED
```

The beginning of the line gives you the file's name. X and Y give the horizontal and vertical location of the cursor. Top indicates which line is the top of your file. Max Lines tells you how many lines you can use for your file. Last represents the last line of your file. If the Last number gets close to the number in Max Lines, you may run out of space. Mem indicates the amount of memory used. The end of the line says if you are in insert or replace mode. Insert and replace is selected by pressing the **Insert** key.

•Cleaning Up the Screen

By pressing **Undo**, the screen will be cleaned up so only your file is displayed. All messages and prompts will be removed.

Appending Files

You can add another file to the end of the file in memory by pressing **F3**. You will be asked to enter of the file you want added. Type the name of the file and press **Return**. The file will be added to the end of the file in memory.

Saving Your File

To save your file, press **F4**. A prompt will appear on the top of the screen, which displays the name of the file. If you want to save an updated version of the file, just press **Return**. If you want to save your file under a different name, type in the new name. TOS names can be up to 8 characters in length, but cannot include spaces. If you want to include a space, use an underline symbol (_) instead. Once you type in the file name, press **Return**.

Δ **Important:** Add a **.c** to the end of your filename to identify your file as a C program.

To work on another file, remove the one in memory by pressing **Shift F4**. A prompt will ask you if you really want to do this. Type **y** and press **Return**, if you do or just press **Return** or answer **n**, if you don't. To read a file into the editor, press **F3**. Type the complete file name at the prompt.

Closing a File

When you are finished with a file, press **F10** to close it. If you have made any changes, you will be asked if you want to save the file. Press **Return**, if you want to save the changes, or **F10** again, if you don't. To quit a file without saving it, press **Shift F10**. You will return to HOS.

Printing a File

You can make a printed listing of your file from HOS. Type **print** followed by the filename. Make sure that your printer is on line and properly connected before printing.

Summary of Ed Commands

KEY	UNSHIFTED	SHIFTED
F1	delete character	delete to end of line
F2	insert character	insert line
F3	read file	replace all
F4	write file	clear file from memory
F5	goto line number	goto marked position
F6	next match	first match
F7	next replace	first replace
F8	copy range to buffer	cut range to buffer
F9	paste buffer	delete range
F10	quit	abort
INSERT	toggle insert/replace	show status
CLR HOME	goto line 1	goto last line
UP ARROW	up line	up page
DOWN ARROW	down line	down page
LEFT ARROW	left character	beginning of line
RIGHT ARROW	right character	end of line
HELP	show this message	
UNDO	refresh screen	
ESC	mark current position as beginning of range	

Chapter 5: The C Compiler

The Haba Hippo-C compiler translates programs written in the C language into 68000 assembly language. The C language is defined in the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie (often referred to as the K&R book). This book is the standard reference guide for the C language.

This chapter assumes you already know C. We recommend you purchase one of the many introductory C books if you are learning C.

How to Start the C Compiler

The C compiler is invoked with the **cc** command. For example:

```
cc  foo.c
c2  foo.i          (2nd pass compile)
```

will compile (cc), then second-pass compile (**c2**) the program foo.c. The status output of the compile will go to directly the screen.

The output of cc is a **.i** file (intermediate) and the **errors** file. The output of the second-pass compile **c2** is the **.s** assembly source file.

After every **cc,** the compiler prints out a line of letters and numbers. This is the compiler's table statistics which shows how full the tables are getting. For example:

H:17/986 N:565/5220 L:7/30 T:1/30 S:2/30 TT:39/248 ST:12/580 E:16/116

These numbers have these meanings:

H:17/986	-Number of symbols	/Max. number of symbols
N:565/5220	-Size of Namelist	/Max. size of Namelist
L:7/30	-Most locals in function	/Max. locals allowed
T:1/30	-Most typedefs active	/Max. typedefs allowed
S:2/30	-Most structures active	/Max. structures allowed
TT:39/248	-Size of Type Table	/Max. size of Type Table
ST:12/580	-Size of Structure Table	/Max. size of Struct. Table
E:16/116	-Number of Globals	/ Max. number of globals

The compiler cannot compile any file who overflows these tables. You can learn to adjust your C program by observing these statistics.

Data Types

The C language allows for several formats of data storage:

char
character

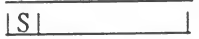
An 8-bit quantity used for ASCII storage or for 8-bit values in the range -128 to 127.

unsigned char

An 8-bit value in the range 0 to 255.

short int

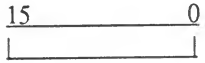
A 16-bit signed integer in the range of -32768 to +32767.



S=sign bit

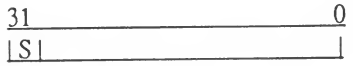
unsigned short int

A 16-bit unsigned integer in the range of 0 to +65536.



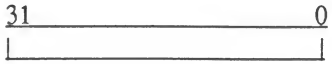
int

A 32-bit signed integer in the range of -2,147,483,648 to +2,147,483,647.

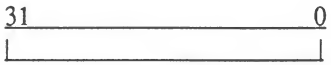


S=sign bit

unsigned int A 32-bit unsigned integer in the range of 0 to +4,294,967,295



pointer A 32-bit integer in the range of 0 to +4,294,967,295



C Operators

The C language provides a set of operators which allow relational expressions, mathematical expressions, and bit control of data.

Unary Operators

* *exp*	indirection
& *lvalue*	pointer to
- *exp*	negate
! *exp*	logical not
~ *exp*	one's complement
++ *lvalue*	prefix increment
-- *lvalue*	prefix decrement
lvalue ++	postfix increment
lvalue --	postfix decrement

Arithmetic Operators

exp + *exp*	addition
exp - *exp*	subtraction
exp * *exp*	multiplication
exp / *exp*	division
exp % *exp*	remainder

Bitwise & Logical Operators

exp & *exp*	bitwise AND
exp | *exp*	bitwise OR
exp ^ *exp*	bitwise exclusive OR
exp << *exp*	shift left
exp >> *exp*	shift right
exp && *exp*	logical AND
exp || *exp*	logical OR

Assignment-expression Operators

lvalue = *exp*

other simple assignment operators are:

+= -= *= /= %= >>= <<= &= ^= |=

In the following table, the rows are in order of decreasing precedence, and items on the same line are of equal precedence.

Table of Precedence

OPERATOR	ASSOCIATIVITY
() [] -> .	left to right
! -- ++ - (TYPE) * & sizeof	right to left
* / %	left to right
+ -	left to right
< <= >= >	left to right
== !=	left to right
&&	left to right
||	left to right
?:	right to left
=	right to left

Calling Assembly Routines from C

It is useful to be able to call assembly routines from C. The following example describes how to interface C programs to assembly routines.

```
/* This explains the Hippo-C to assembly interface */
/*
     Every global C label has an "_" inserted before it.
     Therefore, "main" will become "_main" and "out"
     becomes "_out" at the assembly level.

     To be used in files outside of itself, a label must be
     declared global with the ".global" directive.

     Registers d0,d1,a0,a1 may be used freely, all others
     must be restored to their original state before
     returning.

     The value returned by a function is assumed to be in
     d0.

     Arguments to functions are pushed right to left, all
     values will be long words (4 bytes). After the function
     returns, the caller "cleans up" the stack by adding the
     number of arguments * 4 to the stack pointer.
*/

main()
{
     out("hello,world\n");
}

/* The above file when compiled will become the following
*/
```



```
*************************************************************

     .text                    ;.text places us in the text
                              ; segment
     .global _main            ;.global makes _main a global
                              ; variable
_main:
     link   a6,#.VS           ; allocate space for local
                              ; variables
     movem.l #.MM,-(sp)       ; save the registers we use
     move.l  #S6,-(sp)        ; push address of
"hello,world\n"
     jsr _out                 ; call our assembly function
     move.l  d0,d2            ; save value returned by
                              ; function
     add.l   #4,sp            ; fix up the stack pointer
.L1:
     movem.l (sp)+,#$4        ; restore the registers
     unlk   a6                ; free up space for locals
     rts
     .data                    ; .data places us in the data
                              ; segment
S6:
     .ascii "hello,world\n"
.VS = 0                       ; size of local variables
.MM = $2000                   ; register mask list

; The assembly function _out assumes there is one
; argument on the stack following the return address.
; Therefore to access that argument it is addressed by
; "4(sp)". This function prints the string it is passed and
; returns the number of characters printed.

     .global _out             ; declare _out to be global
_out:
     move.l  4(sp),a0         ; place the argument in a0
     moveq.l #0,d0            ; set index to 0
```

```
outloop:
    move.b  0(a0,d0.l),d1   ; get a character, offset 0 from
a0 + d0
    and.l   #$ff,d1         ; truncate to 8 bits
    beq.s   done            ; done if character == NULL
    move.l  d1,-(sp)        ; push argument to _putchar
    jsr     _putchar        ; character output function
    add.l   #4,sp           ; clean up stack
    addq.l  #1,d0           ; point to next byte in string
    bra     outloop
done:
    rts                     ; d0 has the number of
                            ; characters in the string


;*****************************************************************
;
/*
    Multiple argument example
*/

main()
{
    manyargs(0,1,2,3,4,5,6,7);
}

;*****************************************************************
;
;
; _manyargs will place 8 arguments in the corresponding
; data registers

    .global _manyargs
_manyargs:
    link    a6,#0           ; set a6 to be sp-4 so we can
                            ; use the stack without having
                            ; to keep track of where the
                            ; arguments end up.   Note: the
                            ; "link" opcode pushes the old
                            ; value of a6 on the stack so the
```



```
                            ; first argument is "8(a6)"
    movem.l #$3f00,-(sp)    ; save d2-d7
    move.l  8(a6),d0        ; d0 = 0
    move.l  12(a6),d1       ; d1 = 1
    move.l  16(a6),d2       ; d2 = 2
    move.l  20(a6),d3       ; d3 = 3
    move.l  24(a6),d4       ; d4 = 4
    move.l  28(a6),d5       ; d5 = 5
    move.l  32(a6),d6       ; d6 = 6
    move.l  36(a6),d7       ; d7 = 7

    movem.l (sp)+,#$fc      ; restore d2-d7
    unlk    a6              ; restore a6 and sp
    rts

/* ***************************************************************
    NOTE:
    If a program returns the value 'make', the operating
    system executes the makefile.

    If a program returns a non-zero value to the operating
    system, execution of the makefile stops.

    If a program returns the value 'exec', the operating
    system uses the character string argv[0] as the
    command line

*************************************************************** */
```

Compatibility with Other C Compilers

Haba Hippo-C follows the Kernighan and Ritchie C definition. However, structure passing or returning is not legal. Float or double types are not legal. Finally, all externs should be declared last in local declarations.

Haba Hippo-C Compiler Error Messages

0 - bad abstract declarator in type name cast or sizeof
1 - missing ")" after type name in cast or sizeof
2 - function call is confused
3 - missing "]" in expression
4 - missing lvalue for ++ or --
5 - missing ")" in expression
6 - missing expression for unary operator
7 - missing expression for unary operator
8 - can't use struct member by itself
9 - lvalue needed in expression
10 - functions can't return arrays
11 - need structure
12 - need structure member
13 - need struct member
14 - bad structure member
15 - missing ":" for "?"
16 - missing ";" in declaration
17 - illegal type in declaration
18 - bad type in declaration
19 - too many types in declaration
20 - missing "]" in array declaration
21 - can't take the size of a bit field
22 - bad expression in sizeof
23 - missing "," between declarations
24 - bad declarator in list
25 - can't have function arguments here
26 - need array size
27 - missing "{" in initialization of an aggregate
28 - bad expression in initializer
29 - bad expr in auto initialization
30 - can't initialize automatic aggregates
31 - bad initializer list
32 - bad scalar initialization
33 - missing "}" in scalar initialization
34 - expected "}" in aggregate initialization

35 - illegal expression in initializer
36 - bad constant expression in initializer
37 - undefined variable in expression
38 - too many initializers
39 - can't have arrays of bitfields
40 - can't load arrays
41 - bad expression in "if (BAD)"
42 - in statement following "if (...)"
43 - in statement following "else"
44 - bad expression in "while (BAD)"
45 - in statement following "while (...)"
46 - in statement following "do"
47 - missing "while (" after "do"
48 - bad expression insider "while (BAD)" of "do"
49 - missing ")" or ";" in "do...while (...)"
50 - missing "(" in "for"
51 - bad expression in "for (BAD;...;...)"
52 - bad expression in "for (...;BAD;...)"
53 - bad expression in "for (...;...;BAD)"
54 - bad expression in "for (...;...;...)"
55 - bad expression in "switch (BAD)"
56 - bad statement following "switch (...)"
57 - "case" statement outside of "switch"
58 - bad constant expression in "case BAD:"
59 - sorry, too many cases
60 - missing "-" after constant expression in "case"
61 - bad statement after "case..."
62 - missing ":" after default
63 - "default" outside of "switch"
64 - bad statement after "default;"
65 - missing ";" after "break"
66 - "break" out of place
67 - missing ";" after "continue"
68 - "continue" out of place
69 - bad statement after "return"
70 - missing identifier after "goto"
71 - missing ";" after "goto identifier"
72 - bad expression in statement

73 - missing ";" after expression in statement
74 - missing ":" after label
75 - bad statement following label
76 - missing ";" after expression in statement
77 - if_wh_sw: (
78 - if_wh_sw: expression
79 - EOF encountered during function definition
80 - missing type in struct declaration
81 - can't contain myself
82 - missing "," between struct declarators
83 - bad unnamed bit field size
84 - field too big
85 - bad struct declarator
86 - can't have arguments here
87 - bad bit field constant
88 - field needs non-zero size
89 - field too large
90 - can't find the value
91 - illegal class
92 - variable conflict
93 - conflicting declaration
94 - duplicate global
95 - can't typedef that again
96 - bad declaration
97 - lost an "(" after a function call ?!?
98 - missing ")" after function parameter list
99 - bad init_declaration
100 - missing "{" in function
101 - bad expression in initializer
102 - need to have a 32 bit quantity to initialize with a name
103 - parse error
104 - bad constant expression inside array declaration "[BAD]"
105 - parse error
106 - missing identifier after "&" in initializer
107 - duplicate structure member name

108 - sorry, can't BITFIELD++ or BITFIELD--, use BITFIELD= BITFIELD (+ or -) 1
109 - illegal name in initializer
110 - can't pass structure
111 - floating point not built-in
112 - can't return structure
113 - illegal char in control, line starting with #
114 - bad macro name in #define
115 - bad include filename in #include
116 - can't open file in #include
117 - #include nesting too deep
118 - undefined control: # garbage
119 - missing macro "("
120 - missing macro ")"
121 - probable macro recursion
122 - too much macro nesting
123 - bad macro formal arg in {#define macro(BAD) definintion}

Haba Hippo-C Compiler Warning Messages

The compiler emits warnings when it encounters questionable usage:

0,1 - {pointer + pointer;} This is an undefined construct in C
2 - mismatched types; doint something with a pointer and an integer type (char, short, int, long)
3 - bad subtract operand, {pointer - pointer'} where the two pointer point to different types
4 - indirection operand not a pointer {*expr} expr not a pointer
5 - illegal function
6 - & before struct array ignored
7 - & before array or function ignored
8 - illegal char in control, line starting with #
9 - bad macro name in #define
10 - redefinition of a macro in #define
11 - bad include filename in #include
12 - can't open file in #include
13 - #include nesting too deep
15 - undefined control: # garbage
16 - missing macro "("
17 - missing macro ")"
18 - argument mismatch in a macro
19 - probable macro recursion
20 - too much macro nesting
21 - expression stack overflow
22 - #endif without matching #if
23 - #else without #if
24 - initializer string truncated because of length {char c[1] = "abcde"}
25 - 8 or 9 used in an octal constant
26 - illegal character: "$" or "@" or "`"

27 - bad macro formal arg in (#define macro(BAD) definition)
28 - unknown size of structure or array
29 - can't initialize unions
30 - unknown size of structure or array
31 - compound statement in include file, can't debug
32 - redeclaration of structure
33 - unnamed bit field truncated to fit
34 - mismatched types
35 - not of arithmetic type
36 - function in include file, can't debug
37 - variable not in parameter list of function
38 - function or block in include file, can't debug
39 - can't do that with structures
40 - mismatched return value
41 - type too complicated
42 - macro truncated because of length

Chapter 6: The 68000 Assembler

The 68000 assembler inputs assembly **.s** text source files and outputs machine readable **.o** object modules. The assembly source files can either be generated by the C compiler or can be 68000 assembly code written by the user.

The assembler follows the Motorola opcode mnemonics and supports all of the 68000's addressing modes.

The **asm** (assembler) may be invoked either from HOS in the following form (for example):

```
    asm  foo.s
or  asm  foo.s   foo2.s   foo3.s   foo4.s
```

where **foo.s** is the assembly source.

Numbers

Numbers are assumed to be decimal. Numbers may be defined in hex by using the $ character in front of the number (e.g. $10=16). Decimal numbers may be preceded by a minus (-) sign.

Labels

The Haba Hippo-C 68000 assembler accepts labels of practically any length. Valid label characters are a-z, A-Z, 0-9, and the _ character. Each label must start with either a-z, A-Z, or the _ character. Local labels start with a period character. Upper case characters are differentiated from lower case characters (a≠A). Each label is aligned to start on an even address. To force PC relative addressing mode, use the format label^(pc). **Each label must start in column 1**, and can be optionally followed by a colon.

Local labels always start with a period (.) in column one. Local labels are "local" between two non-local labels.

Expressions

The assembler allows very limited expressions. The only expression allowed is exactly:

```
    label+n       (where n is a number)
or  label+-n      (label minus n)
```

Directives

Directives tell the assembler what to do, and are not part of the 68000 opcode set. All directives may start in any column **except column one** (otherwise the assembler would interpret it as a label). The Hippo assembler allows these directives:

.byte
Places the specified bytes immediately into the code. Allows multiple numbers to be defined.
(e.g. .byte 8,9,10,11,$ff)

.word
Places the specified words (2 bytes each) immediately into the code. Allows multiple numbers to be defined.
(e.g. .word 8,9,10,11,$ff)

.long
Places the specified long words (4 bytes each) immediately into the code. Allows multiple numbers to be defined.
(e.g. .long 8,9,10,11,$ff)

.ascii
Places the specified characters immediately into the code. Only one string can be defined. If the string is in double quotes (e.g. .ascii "Hello world\n"), then the string will be in C form (null terminated). If the string is in single quotes, then the string will be in Pascal form (data prefixed with the character count). Two additional characters are accepted within the strings: \n (same as a CR,LF), \r (just a CR - allows overprinting).

.even
Aligns to an even byte address.

.text
Specifies that the following belongs in the code segment

.data
Specifies that the following belongs in the data segment

.bss
Specifies that the following belongs in the variable space segment. The only directive allowed within this segment is the .space directive.

.space n
Specifies to allocate n number of bytes and to initialize them to zero.

.global label_name

Specifies that the label label_name is to be global and can be accessed by other modules. .global is sometimes called .extern in other assemblers. The declaration that the label is global (the .global directive) must appear before the label is actually defined. We recommend that you use globals in this form:

```
.global label_name   ; specifies that the label is global
label_name:          ; defines actual position of label
```


Chapter 7: The Linker

The Haba Hippo-C linker combines many small object modules (**.o** files) produced by the assembler into one large executable **a.out** file. The a.out file may then be executed directly (by typing **a.out** in HOS). You can also use the linker to create stand-alone application which can be opened from the Desktop.

How to Use the Linker

The linker is invoked with the **ld** command. For example:

```
ld  foo.o
```

will link the object file **foo.o** with standard C library files, remove **foo.i** and **foo.s** files, and produce an executable file **a.out**.

Multiple **.o** object files can be linked together by simply listing the series of .o files like this:

```
ld    foo1.o  foo2.o  foo3.o  foo4.o
```

A maximum of approximately 50 object files can be linked together.

You may also link files from disks other than the default drive by preceding the disk name with the volume name and a colon like this:

```
ld    b:\subdir\foo.o
```

The Library

The linker draws from a variety of files to collect all of the C functions called from the **.o** object files. This list of C functions is in the file **library.dir**. You can **cat library.dir** to see all of the standard modules that are available.

You can modify **library.dir** by using the **ar**(archive) command.

The ar (Archive) Command

The **ar** command creates the **library.dir** file based on input from the file **archive.list**. The **library.dir** file is used by the **ld** (linker) command as a list of available routines to link. The **archive.list** file typically contains these modules:

```
aes.o
asm.o
lib.o
printf.o
st.o
stdio.o
vdi.o
```

You can, of course, add your own modules to **archive.list**, then execute the **ar** command like this:

```
ar
```

This will then create the **library.dir** file for you.

Stand-alone Applications

Programs which run directly under HOS have a slightly different header than programs intended to run directly under GEM. If you want your application to become double-clickable from GEM, then you should invoke the **-S** option when linking. When linking with the **-S** option, the linker produces the executable file **a.prg** at the top level (i.e. **\a.prg**) instead of the usual **a.out** in the current directory. For example:

```
ld -S foo.o
```

Chapter 8: GEMDOS

The Atari ST Hippo-C supports the following GEMDOS routines:

ROUTINE	WHAT IT DOES
term0()	exit to desktop
conin()	input character from keyboard
conout(c)	output character c to screen
auxin()	input character from serial port
auxout(c)	output character to serial port
prnout(c)	output character to printer
rawio(c)	if c == 0xff input character, else output c
rawcin()	input character with no echo, no control chars
necin()	input character with no echo, contrôl chars work
conws(s)	prints string s to screen
conrs(buf)	buf[0] = length of buf, buf[1] = chars read, buf[2] thru n are bytes read from keyboard
conis()	returns -1 if keyboard character available, else 0
setdrv(newdrv)	0=A,1=B,... Makes newdrv current drive, returns drive map of system (bit0=A,bit1=B...)
conos()	-1 if screen is ready for a character, else 0
prnos()	-1 if printer is ready for a character, else 0
auxis()	-1 if character available from serial port, else 0
auxos()	-1 if serial port is ready for character, else 0
getdrv()	returns code of current drive, 0=A,1=B ...15=P
setdta(p)	sets disk transfer address used by sfirst to p
getdate()	bits 0-4=day, 5-8=month, 9-15=years since 1980
setdate(date)	same format as getdate()

ROUTINE	WHAT IT DOES
gettime()	bits 0-4=2 seconds, 5-10=minutes, 11-15=hours
settime(time)	same format as gettime()
getdta()	returns disk transfer address used by sfirst()
version()	returns version, byte1=minor, byte0=major
d_free(buf,drive)	0=default,1=A,... struct { int free,cpd,bps,spc }buf
d_create(pathname)	make a sub-directory, 0 if success, else non-zero
d_delete(pathname)	delete a sub-directory,0 if success,else non-zero
setpath(pathname)	change directory, 0 if success, non-zero if not
create(name,attr)	attr:1=R/O,2=hidden,4=system,8=vol, <0 if error
open(name,mode)	mode:0=R/O,1=W/O,2=R/W, <0 if error else handle
close(handle)	0 if success, else non-zero
read(handle,count,buf)	returns number of characters read, <0 if error
write(handle,count,buf)	returns number of characters written, <0 if error
delete(name)	deletes file with name
seek(delta,handle,mode)	mode:0=absolute,1=relative,2=from EOF
attrib(name,wrt,mode)	wrt:0=get/1=set, mode:1=R/O, 2=hidden, 4=system, 8=volumename, 16=sub-directory, 32=closed
getpath(buf,drive)	0=default,1=A,... puts current pathname in buf[64]
alloc(n),malloc(n)	allocate n bytes of memory
free(p)	free memory from malloc() or alloc()
shrink(start,size)	returns size bytes from start to GEMDOS
sfirst(pspec,attr)	pspec=name with wildcarding, attr: see attrib(), DTA: 0-20 reserved, 21:attr, 22-23:time, 24-25:date, 26-29:size, 30-43:name
snext()	next file in wildcard expansion, 0 if found
rename(old,new)	rename old to new

ROUTINE	WHAT IT DOES
get_mpb(buf)	get memory parameter block
char_input_status(h)	h:0=PRN,1=AUX,2=CON, -1 device ready, 0 if not
char_input(h)	h:same as char_input_status(h), but waits until character is input
char_output(h,char)	h:same as char_input, but outputs character to device h
read_write_sectors(wrt,buf,num,start,drive)	wrt:0=R,1=W num=sectors
set_exception_vector(vecnum,vecaddr)	
get_timer_ticks()	nearest millisecond
get_bpd(drive)	get BIOS parameter block for drive 0=A,....
char_output_status(h)	same as char_input_status, but h is -1 if device is ready, 0 if not
media_change(drive)	0 if not, 1 if possible, 2 if changed
get_drive_map()	bit0=A,....
get_shift_cont(flag)	bit0: RSHF, 1:LSHF, 2=ctrl, 3=alt, 4=lock, -1=read only

Chapter 9: VDI and AES bindings

Haba Hippo-C supports the following VDI routines:

v_opnwk(work_in, handle, work_out)
v_clswk(handle)
v_opnvwk(work_in, handle, work_out)
v_clsvwk(handle)
v_clrwk(handle)
v_updwk(handle)
vst_load_fonts(handle, select)
vs_clip(handle, clip_flag, xy)
v_pline(handle, count, xy)
v_pmarker(handle, count, xy)
v_gtext(handle, x, y, string)
v_fillarea(handle, count, xy)
v_cellarray(handle, xy, row_length, el_per_row, num_rows, wr_mode, colors)
v_contourfill(handle, x, y, index)
vr_recfl(handle, xy)
v_bar(handle, xy)
v_arc(handle, xc, yc, rad, sang, eang)
v_pieslice(handle, xc, yc, rad, sang, eang)
v_circle(handle, xc, yc, rad)
v_ellarc(handle, xc, yc, xrad, yrad, sang, eang)
v_ellpie(handle, xc, yc, xrad, yrad, sang, eang)
v_ellipse(handle, xc, yc, xrad, yrad)
v_rbox(handle, xy)
v_rfbox(handle, xy)
v_justified(handle, x, y, string, length, word_space, char_space)
vswr_mode(handle, mode)
vs_color(handle, index, rgb)
vsl_type(handle, style)
vsl_udsty(handle, pattern)
vsl_width(handle, width)
vsl_color(handle, index)
vsl_ends(handle, beg_style, end_style)
vsm_type(handle, symbol)
vsm_height(handle, height)

vsm_color(handle, index)
vst_height(handle, height, char_width, char_height, cell_width,
cell_height)
vst_point(handle, point, char_width, char_height, cell_width,
cell_height)
vst_rotation(handle, angle)
vst_font(handle, font)
vst_color(handle, index)
vst_effects(handle, effect)
vst_alignment(handle, hor_in, vert_in, hor_out, vert_out)
vsf_interior(handle, style)
vsf_style(handle, index)
vsf_color(handle, index)
vsf_perimeter(handle, per_vis)
vsf_udpat(handle, fill_pat, planes)
vro_cpyfm(handle, wr_mode, xy, srcMFDB, desMFDB)
vrt_cpyfm(handle, wr_mode, xy, srcMFDB, desMFDB, index)
vr_trnfm(handle, srcMFDB, desMFDB)
v_get_pixel(handle, x, y, pel, index)
vsin_mode(handle, dev_type, mode)
vrq_locator(handle, initx, inity, xout, yout, term)
vsm_locator(handle, initx, inity, xout, yout, term)
vrq_valuator(handle, val_in, val_out, term)
vsm_valuator(handle, val_in, val_out, term, status)
vrq_choice(handle, in_choice, out_choice)
vsm_choice(handle, choice)
vrq_string(handle, length, echo_mode, echo_xy, string)
vsm_string(handle, length, echo_mode, echo_xy, string)
vsc_form(handle, cur_form)
vex_timv(handle, tim_addr, old_addr, scale)
v_show_c(handle, reset)
v_hide_c(handle)
vq_mouse(handle, status, px, py)
vex_butv(handle, usercode, savecode)
vex_motv(handle, usercode, savecode)
vex_curv(handle, usercode, savecode)
vq_key_s(handle, status)
vq_extnd(handle, owflag, work_out)
vq_color(handle, index, set_flag, rgb)
vql_attributes(handle, attributes)

vqm_attributes(handle, attributes)
vqf_attributes(handle, attributes)
vqt_attributes(handle, attributes)
vqt_extent(handle, string, extent)
vqt_width(handle, character, cell_width, left_delta, right_delta)
vqt_name(handle, element_num, name)
vq_cellarray(handle, xy, row_len, num_rows, el_used,
rows_used, stat, colors)
vqin_mode(handle, dev_type, mode)
vqt_font_info(handle, minADE, maxADE, distances, maxwidth,
effects)
vq_chcells(handle, rows, columns)
v_exit_cur(handle)
v_enter_cur(handle)
v_currup(handle)
v_curdown(handle)
v_curright(handle)
v_curleft(handle)
v_curhome (handle)
v_eeos(handle)
v_eeol(handle)
vs_curaddress(handle, row, column)
v_curtext(handle, string)
v_rvon(handle)
v_rvoff(handle)
vq_curaddress(handle, row, column)
vq_tabstatus(handle)
v_hardcopy(handle)
v_dspcur(handle, x, y)
v_rmcur(handle)
v_form_adv(handle)
v_output_window(handle, xy)
v_clear_disp_list(handle)
v_bit_image(handle, filename, aspect, scaling, num_pts, xy)
vs_palette(handle, palette)
vqp_films(handle, names)
vqp_state(handle, port, filmnum, lightness, interlace, planes,
indexes)
vsp_state(handle, port, filmnum, lightness, interlace, planes,
indexes)

vsp_save(handle)
vsp_message(handle)
vqp_error(handle)
v_meta_extents(handle, min_x, min_y, max_x, max_y)
v_write_meta(handle, num_ints, ints, num_pts, pts)
vm_filename(handle, filename)

Haba Hippo-C supports the following AES routines:

appl_init()
appl_exit()
appl_write(rwid, length, pbuff)
appl_read(rwid, length, pbuff)
appl_find(pname)
appl_tplay(tbuffer, tlength, tscale)
appl_trecord(tbuffer, tlength)
evnt_keybd()
evnt_button(clicks, mask, state, pmx, pmy, pmb, pks)
evnt_mouse(flags, x, y, width, height, pmx, pmy, pmb, pks)
evnt_mesag(pbuff)
evnt_timer(locnt, hicnt)
evnt_multi(flags, bclk, bmsk, bst, mlflags, mix, miy, miw,
m1h, m2flags, m2x, m2y, m2w, m2h, mepbuff, tlc, thc, pmx,
pmy, pmb, pks, pkr, pbr)
evnt_dclick(rate, setit)
menu_bar(tree, showit)
menu_icheck(tree, itemnum, checkit)
menu_ienable(tre, itemnum, enableit)
menu_tnormal(tree, titlenum, normalit)
menu_text(tree, inum, ptext)
menu_register(pid, pstr)
objc_add(tree, parent, child)
objc_delete(tree, delob)
objc_draw(tree, drawob, depth, xc, yc, wc, hc)
objc_find(tree, startob, depth, mx, my)
objc_order(treee, mov_obj, newpos)
objc_offset(tree, obj, poffx, poffy)
objc_edit(tree, obj, inchar, idx, kind)
objc_change(tree, drawob, depth, mx, yc, wc, hc, newstate,
redraw)
form_do(form, start)
form_alert(defbut, astring)
form_error(errnum)
form_center(tree, pcx, pcy, pcw, pch)
graf_handle(pwchar, phchar, pwbox, phbox)
graf_mouse(m_number, m_addr)
graf_mkstate(pmx, pmy, pmstate, pkstate)

scrp_read(pscrap)
scrp_write(pscrap)
fsel_input(pipath, pisel, pbutton)
wind_create(kind, wx, wy, ww, wh)
wind_open(handle, wx, wy, ww, wh)
wind_close(handle)
wind_delete(handle)
wind_get(w_handle, w_field, pw1, pw2, pw3, pw4)
wind_set(w_handle, w_field, w2, w3, w4, w5)
wind_find(mx, my)
wind_update(beg_update)
wind_calc(wctype, kind, x, y, w, h, px, py, pw, ph)
rsrc_load(rsname)
rsrc_free()
rsrc_gaddr(rstype, rsid, paddr)
rsrc_saddr(rstype, rsid, lngval)
rsrc_abfix(tree, obj)
shel_read(pcmd, ptail)
shel_write(doex, isgr, iscr, pcmd, ptail)
shel_get(pbuffer, len)
shel_put(pdata, len)
shel_find(ppath)
shel_envrn(ppath, psrch)

Appendix: Sample Programs

Your Haba Hippo-C comes with sample programs that you may use as examples. These programs show you how to use some of the Atari ST's features. They are located in the \USR\ subdirectory and include:

APPLRW.C
DOWN.C
FILE.C
KEYBOARD.C
MOUSE.C
SUM.C
WINDOW.C
HELLO.BAT

These programs are uncompiled, so you can examine them in the editor. You can compile them with Haba Hippo-C's compiler.

We hope you will enjoy using Haba Hippo-C.